An Implementation of Fast Agglomerative Hierarchical Clustering Algorithm Using Locality-Sensitive Hashing by Walker Harrison and Lisa Lebovici

May 2, 2018

1 Abstract

Agglomerative hierarchical clustering is a widely used clustering algorithm in which data points are sequentially merged according to a distance metric from individual clusters into a single cluster. While many different metrics exist to evaluate inter-cluster distance, single-linkage is among the most popular, in part due to its simplicity. According to this method, pairwise distances are computed between all points, and the clusters containing the two points with the shortest Euclidean distance are joined in each iteration. However, single-linkage has some drawbacks, most notably that it scales quadratically as data grows, since each additional point must be measured from all of the existing ones. As such, Koga, Ishibashi, and Watanbe propose using locality-sensitive hashing (hereafter LSH) as an approximation to the single-linkage method. Under LSH, points are hashed into buckets such that the distance from a given point p need only be computed to the subset of points with which it shares a bucket.

In this paper, we implement LSH as a linkage method for agglomerative hierarchical clustering, optimize the code with an emphasis on reducing the time complexity of the creation of the hash tables, and apply the algorithm to a variety of synthetic and real-world datasets.

Keywords: agglomerative hierarchical clustering, locality-sensitive hashing, single-linkage, nearest neighbor search, dendrogram similarity, cophenetic correlation coefficient, unsupervised learning

Github link: https://github.com/lisalebovici/LSHLinkClustering

2 Background

As data in the modern age gets easier and easier to quickly accumulate, techniques that can accurately segment or organize it are becoming increasingly important. In particular, unsupervised learning — that is, data which has no "ground truth" representation — finds itself at the center of many fields, ranging from consumer marketing to computer vision to genetics. At its heart, the goal of unsupervised learning is pattern recognition. To this end, algorithms such as k-means and agglomerative hierarchical clustering have been sufficiently effective until now; but as the size of data continues to grow, scalability issues are becoming a bigger barrier to analysis and understanding.

Koga et al.'s *Fast Agglomerative Hierarchical Clustering Algorithm Using Locality-Sensitive Hashing* provides a faster approximation to single-linkage hierarchical agglomerative clustering for large data, which has a run time complexity of $O(n^2)$. The single-linkage method requires that the distances from every point to all other points be calculated, which becomes prohibitively expensive. In contrast, the primary advantage of LSH is that it reduces the number of distances that need to be computed as well as the the number of iterations that need to run, resulting in linear run time complexity $O(nB)$ (where B is the maximum number of points in a single hash table).

However, one consequence of the gain in run time is that LSH results in a coarser approximation of the data than single-linkage. More precisely, estimating which points are close with a hash function instead of explicitly examining each candidate comes at the cost of potentially overlooking nearby points. Plus, since all clusters within a threshold distance r are merged, LSH by definition has many fewer iterations than single-linkage. This granularity can be controlled by a parameter within the algorithm, but in general LSH is not expected to reproduce the single-linkage method exactly. Nonetheless, when granularity is not a primary concern, the improvements in terms of efficiency make it a worthwhile alternative.

We will proceed by describing the implementation of LSH for hierarchical clustering.

3 Description of Algorithm

Some important notation will first be defined:

- n: number of rows in data
- d: dimension of data
- C: least integer greater than the maximal coordinate value in the data
- ℓ: number of hash functions
- k: number of sampled bits from a hashed value
- r: minimal distance between points required to merge clusters
- A: increase ratio of r on each iteration

LSH works in two primary phases: first, by creating a series of hash tables in which the data points are placed, and second, by computing the distances between a point and its "similar" points, as defined by sharing a hash bucket, to determine which clusters should be merged.

Phase 1: Generation of hash tables. Suppose we have a d-dimensional point x. A unary function is applied to each coordinate value of x such that x_i 1s are followed by $C - x_i$ 0s. The sequence of 1s and 0s for all coordinate values of x are then joined to form the hashed point of x. Some number k bits are then randomly sampled without replacement from the hashed point. For example, if x is the point $(2, 1, 3)$ and $C = 4$, the hashed point would be $\underbrace{1100}_{2}\underbrace{1000}_{1}\underbrace{1110}_{3}$. If $k = 5$ and the indices $I = 5, 9, 2, 11, 8$ are randomly sampled, then our resulting value would be 11110. x would thus be placed into the hash table for 11110.

This hash function is applied to all points, and a point is added to the corresponding hash table if no other point in its cluster is already present. Another set of k indices I is then randomly sampled and the resulting hash function is applied to all of the points. This procedure is repeated ℓ times.

The intuition behind this step is as follows: a point s that is very similar to x is likely to have a similar hash sequence to x and therefore appear in many of the same hash tables as x. However, for any given hash, there is no guarantee; for example, for $s = (1, 1, 3)$, the hashed point would be $\underbrace{1000}_{1}\underbrace{1000}_{1}\underbrace{1110}_{3}$ and the sampled value would be 11010. In this case, s would not be in the same bucket as x above. However, by applying ℓ hash functions to the data, we have some certainty that if s really is close to x, it will share at least one hash table.

Phase 2: Nearest neighbor search. For each point x, we find all of the points that share at least one hash table with x and are not currently in the same cluster as x. These are the similar points which are candidates to have their clusters merged with x's cluster. The distances between x and the similar points are computed, and for any point p for which the Euclidean distance between x and p is less than r, p's cluster is merged with x's cluster.

If there is more than one cluster remaining after the merges, the values for r and k are updated and then phases 1 and 2 are repeated. r is increased (we now consider points that are slightly further away than previously to merge more distant clusters) and k is decreased (so that the hashed values become shorter and therefore more common). This continues until all points are in the same cluster, at which point the algorithm terminates.

LSH-Link Algorithm

```
    Input: Starting values for $\ell$, $k$, and $A$
    Initialize:
        if $n < 500$
            sample $M = \{\sqrt{n}$ points from data$\}$
            $r = \min \text{dist}(p, q)$, where $p, q \in M$
        else
            $r = \frac{d*C}{2*(k+d)}\sqrt{d}$
    while num_clusters > 1:
        for $i = 1, .., \ell$:
            $unary_C(x) = \underbrace{11...11}_{x}\underbrace{00...00}_{C-x}$
            sample $k$ bits from $unary_C(x)$
            if $x$'s cluster is not in hash table:
                add $x$ to hash table
            repeat for $n$ points
        for $p = 1, .., n$:
            S = {set of points that share at least one hash table with $p$}
            Q = {$q \in S$ s.t. $\text{dist}(p, q) < r$}
            merge $Q$'s clusters with $p$'s cluster
        if num_clusters > 1:
            $r = A * r$
            $k = \frac{d*C}{2*r}\sqrt{d}$
```

4 Performance Optimizations

Run time profiling for `build_hash_tables()` and `LSHLink()` are below, as well as a comparison to `scipy`'s single-linkage hierarchical clustering and our own implementation of agglomerative hierarchical clustering.

The three different functions were applied to an extended version of the *iris* dataset; we built a set of 1,500 points based on the original 150 points with random noise added. As seen below, `scipy`'s implementation is clearly the superior method, running in an impressive 16.7 ms. Note that this is to be expected since a great deal of it is written in C.

In comparison, our implementation of single-linkage hierarchical clustering, written purely in Python, ran in 2 minutes 46 seconds, while the LSH version ran in 2 minutes 13 seconds. A plot in the *Applications to Simulated Datasets* section below will show that the speed advantage for LSH vs. single-linkage does not necessarily hold for smaller datasets.

Note from the profiling below that the vast majority of the run time in LSH is spent in `build_hash_tables()`; the code for nearest neighbor search and merging clusters is relatively trivial. The results show that of the 157 seconds spent running `LSHLink()`, 151 seconds were in `build_hash_tables()`. As such, nearly all of the optimization efforts were spent in speeding up the creation of the hash tables.

```
In [16]: from imports import *
         import v1
         import funcs
```

```
In [14]: iris = datasets.load_iris().data * 10
         iris_big = datasets.load_iris().data
         iris_big = v1.data_extend(iris_big, 10) * 10
         iris_big += np.abs(np.min(iris_big))
```

```
In [15]: %timeit -r1 linkage(iris_big, method = 'single')
```

```
19.3 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 10 loops each)
```

```
In [21]: %timeit -r1 lsh.singleLink(1, iris_big)
```

```
2min 46s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [6]: %timeit -r1 v1.LSHLinkv1(iris_big, A = 1.4, l = 10, k = 100)
```

```
2min 13s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [7]: %prun -q -D LSHLink.prof v1.LSHLinkv1(iris_big, A = 1.4, l = 10, k = 100)
```

```
*** Profile stats marshalled to file 'LSHLink.prof'.
```

```
In [15]: p = pstats.Stats('LSHLink.prof')
         p.sort_stats('time', 'cumulative').print_stats(
             'build_hash_tables')
         p.sort_stats('time', 'cumulative').print_stats(
             'LSHLink')
         pass

Mon Apr 30 14:18:31 2018    LSHLink.prof

         26039956 function calls in 157.957 seconds

   Ordered by: internal time, cumulative time
   List reduced from 60 to 1 due to restriction <'build_hash_tables'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        7   11.719    1.674  151.201   21.600 <ipython-input-1-b260ed9440e0>:68(build_hash_tab


Mon Apr 30 14:18:31 2018    LSHLink.prof

         26039956 function calls in 157.957 seconds

   Ordered by: internal time, cumulative time
   List reduced from 60 to 1 due to restriction <'LSHLink'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.689    0.689  157.956  157.956 <ipython-input-1-b260ed9440e0>:93(LSHLink)
```

The primary optimization in the second version of the code was caching the hash points using `lru_cache()` from `functools`. By storing the hashed unary values and point representations, `build_hash_tables()` only had to compute each point's hash value on the initial run, rather than on each iteration; this was possible since a point's unary representation remains the same for a given C value, and C does not change throughout the course of the algorithm.

The other optimization was to introduce more control logic into the nearest neighbor search. Specifically, additional "if" statements were added, such that if no points were found within distance r of point p, that iteration of the `for` loop terminates rather than continuing on to attempt to find clusters that can be merged.

These changes reduced the run time of LSH to around 20 seconds in our tests on the *iris* version with 1500 points. Run times are shown below.

```
In [11]: lsh.clear_caches()
         %timeit -r1 lsh.LSHLink(
             iris_big, A = 1.4, l = 10, k = 100, seed1 = 12, seed2 = 6)

22.9 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

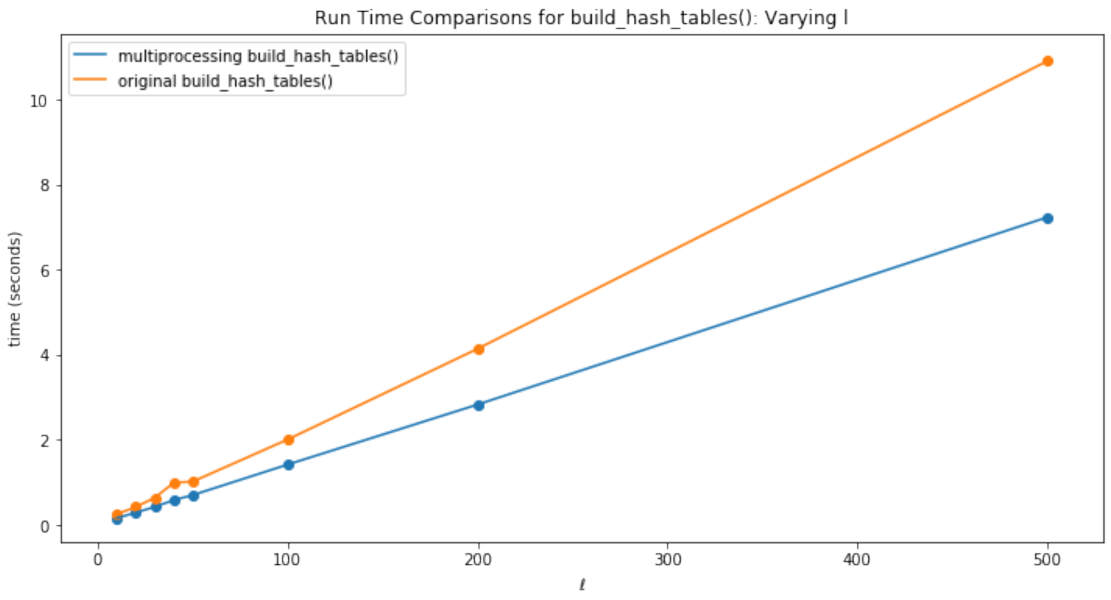
```
In [40]: lsh.clear_caches()
         %prun -q -D LSHLink2.prof lsh.LSHLink(
             iris_big, A = 1.4, l = 10, k = 100, seed1 = 12, seed2 = 6)


*** Profile stats marshalled to file 'LSHLink2.prof'.


In [41]: p = pstats.Stats('LSHLink2.prof')
         p.sort_stats('time', 'cumulative').print_stats(
             'build_hash_tables')
         p.sort_stats('time', 'cumulative').print_stats(
             'LSHLink')
         pass

Mon Apr 30 17:11:59 2018    LSHLink2.prof

         452355 function calls in 20.685 seconds

   Ordered by: internal time, cumulative time
   List reduced from 45 to 1 due to restriction <'build_hash_tables'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        6   16.995    2.832   18.909    3.152 <ipython-input-37-8c500c6efc8a>:87(build_hash_ta


Mon Apr 30 17:11:59 2018    LSHLink2.prof

         452355 function calls in 20.685 seconds

   Ordered by: internal time, cumulative time
   List reduced from 45 to 1 due to restriction <'LSHLink'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.215    1.215   20.684   20.684 <ipython-input-37-8c500c6efc8a>:116(LSHLink)
```

Since our `build_hash_tables()` function includes a loop to create the various hash tables (ℓ hash tables, to be precise), it's a good candidate for multiprocessing. More specifically, if we create each of the hash tables in parallel, the total time to run should be reduced.

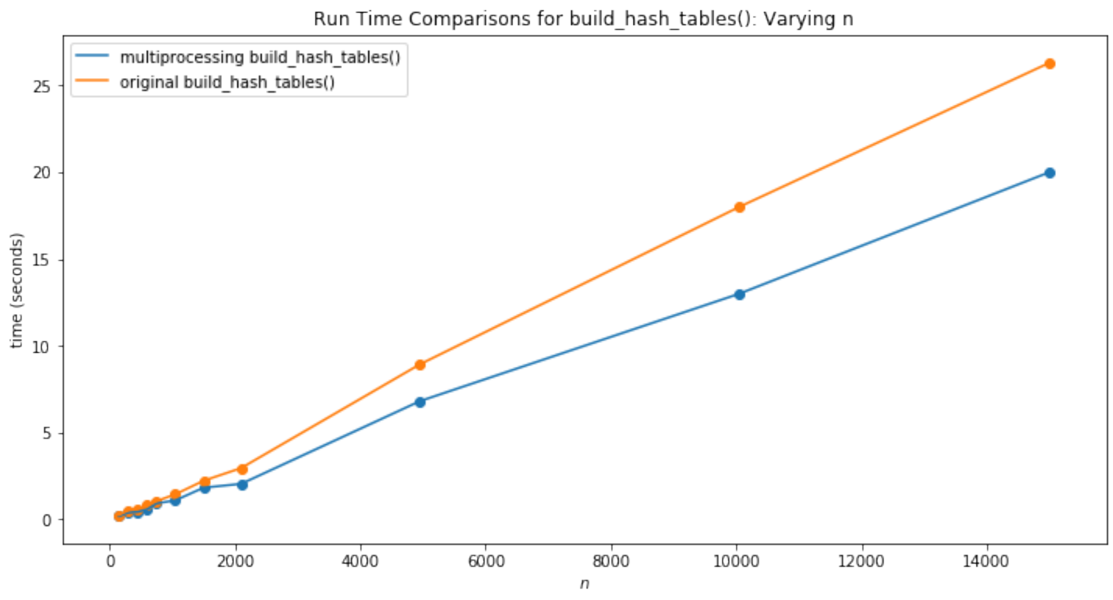
Of course, for these tasks to be truly parallel there can't be any interdependencies, which is not the case in our original `build_hash_tables()` function. Each hash function updates a shared dictionary, which would slow down or completely derail an attempt at parallel execution since each competing hash table would lock the dictionary from the others when updating. So to utilize multiprocessing, we had to redefine the workhorse function as `build_hash_table()` so that it created and returned its own individual dictionary, which can then be zipped with the others into a single hash table.

There is overhead associated with multiprocessing so that there are no or minimal efficiency gains seen for small datasets with limited numbers of hash tables (small ℓ in other words). But as you increase ℓ or n, multiprocessing becomes more and more useful, allowing the parallel version of the function to eventually outperform the original. We illustrate this below and the raw data can be found in the Appendix.

```
In [17]: mp1_x = [10, 20, 30, 40, 50, 100, 200, 500]
         mp1_y = [.162, .291, .426, .586, .703, 1.42, 2.83, 7.23]

         bht1_x = [10, 20, 30, 40, 50, 100, 200, 500]
         bht1_y = [.256, .423, .641, .993, 1.02, 2.01, 4.14, 10.9]

         funcs.mp_run_times(mp1_x, mp1_y, bht1_x, bht1_y,
                      title = 'Run Time Comparisons for build_hash_tables(): Varying l',
                      xlabel = r'$\ell$',
                      ylabel = 'time (seconds)')
```



```
In [18]: mp2_x = [150, 300, 450, 600, 750, 1050, 1500, 2100, 4950, 10050, 15000]
         mp2_y = [.181, .391, .440, .577, .925, 1.1, 1.83, 2.05, 6.81, 13, 20]

         bht2_x = [150, 300, 450, 600, 750, 1050, 1500, 2100, 4950, 10050, 15000]
         bht2_y = [.257, .530, .624, .821, 1.06, 1.45, 2.23, 2.96, 8.93, 18, 26.3]

         funcs.mp_run_times(mp2_x, mp2_y, bht2_x, bht2_y,
                      title = 'Run Time Comparisons for build_hash_tables(): Varying n',
                      xlabel = r'$n$',
                      ylabel = 'time (seconds)')
```



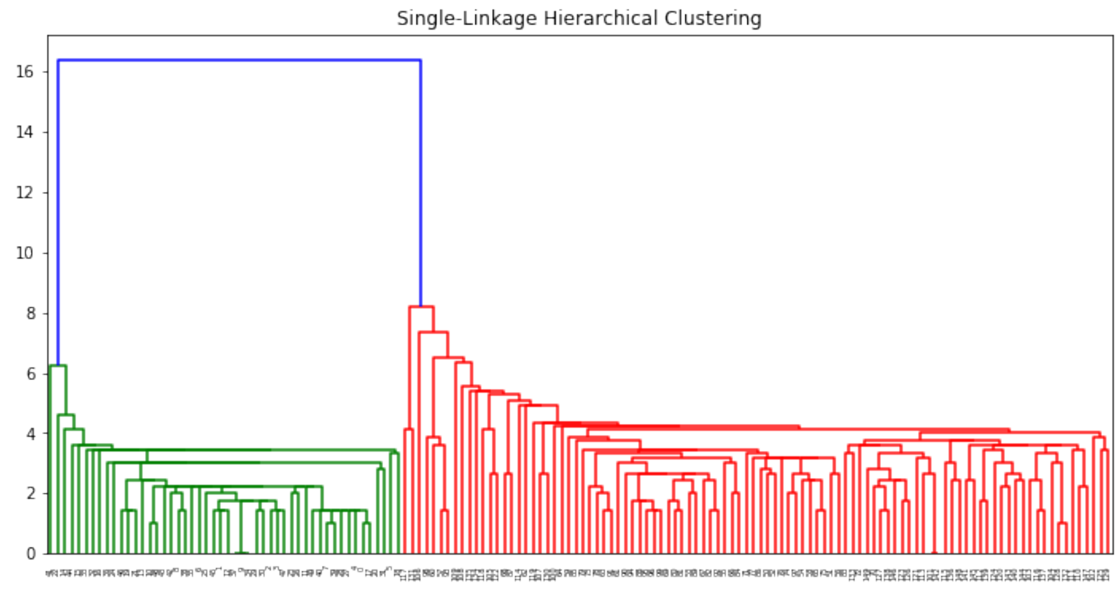
That being said, multiprocessing ultimately didn't improve the timing nearly as much as simply caching the unary representation of points (i.e. the bitstream pre-sampling). That fact, combined with technical difficulties associated with marrying the two approaches, led us to only include the cached version in the final package without any parallelization.

5 Applications to Simulated Datasets

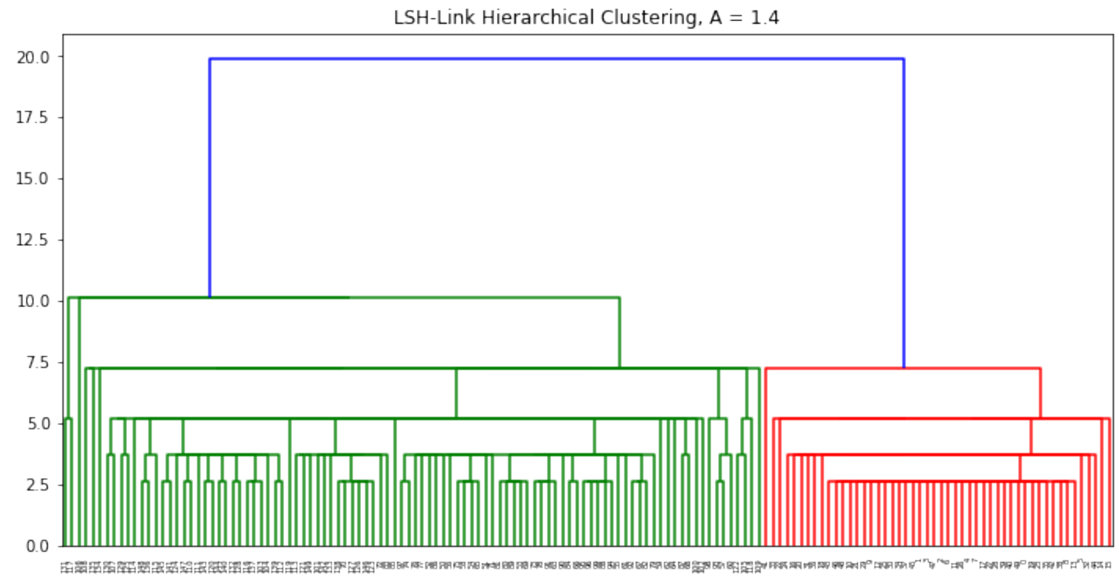
5.0.1 Iris Dataset

The algorithm was first applied to the *iris* dataset, as in the paper by Koga et al., in an effort to reproduce their results. Below, we compare the dendrograms produced from running single-linkage hierarchical clustering with our implementation of LSH under two slightly different parameterizations. In both instances, we set initial values for k and ℓ to be 100 and 10, respectively. However, the first run of LSH has the increase ratio $A = 1.4$ while the second run has $A = 2.0$.

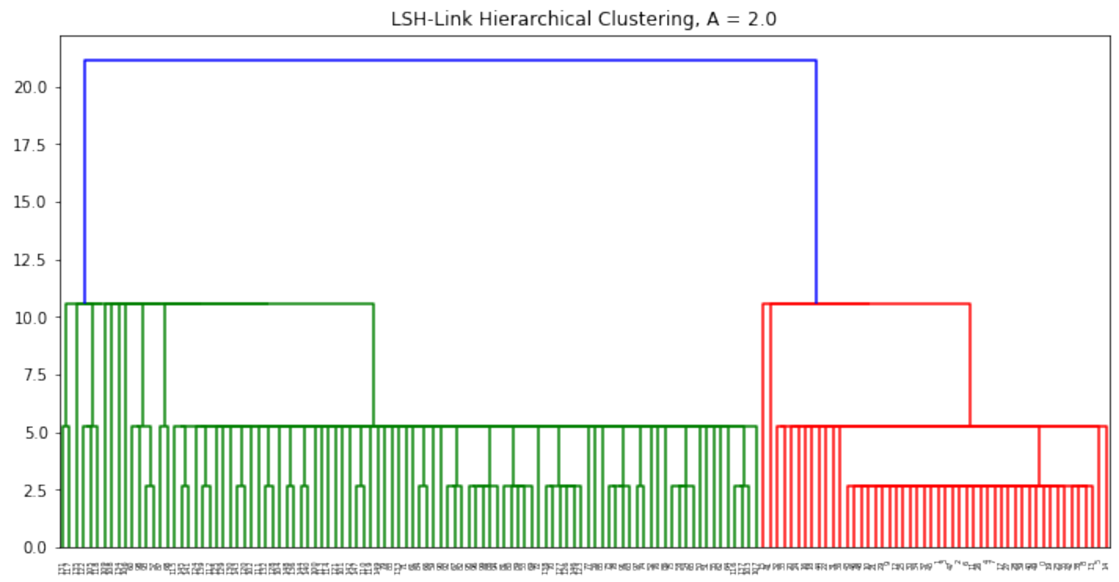
```
In [138]: z = linkage(iris, method="single")

In [139]: dendrogram(z)
          plt.gcf().set_size_inches(12, 6)
          plt.title('Single-Linkage Hierarchical Clustering')
          plt.show();
```



```
In [140]: clusters, Z = lsh.LSHLink(iris, A = 1.4, l = 10, k = 100,
                                    dendrogram = True, seed1 = 12, seed2 = 6)

In [141]: dendrogram(Z, color_threshold = 18)
          plt.gcf().set_size_inches(12, 6)
          plt.title('LSH-Link Hierarchical Clustering, A = 1.4')
          plt.show();
```



```
In [142]: clusters2, Z2 = lsh.LSHLink(iris, A = 2.0, l = 10, k = 100,
                                      dendrogram = True, seed1 = 12, seed2 = 6)

In [143]: dendrogram(Z2, color_threshold = 18)
          plt.gcf().set_size_inches(12, 6)
          plt.title('LSH-Link Hierarchical Clustering, A = 2.0')
          plt.show();
```



As in the paper, we have confirmed that the members of the top two clusters match perfectly across all three implementations (note that the green cluster in the first figure above is equivalent to the red clusters in the second and third figures). However, as expected, the increase in A corresponds to a sparser dendrogram; the algorithm is now making bigger leaps in r with each iteration, so it will merge a greater number of clusters at a given time. The height of the dendrogram shows us the point in time at which two clusters merged. For single-linkage, we see that there are n-1 unique merge times, which is indeed the behavior of the single-linkage method. Under LSH with $A = 1.4$, there are six iterations of cluster merges, while under LSH with $A = 2.0$ there are only four iterations until the data is fully merged. If a high level of preciseness is not needed, however, the graphs above show that LSH does give us a rough approximation of single-linkage.

One way to measure the similarity between two dendrograms is called cophenetic correlation. For two dendograms X and Y it is given by

$$r_{X,Y} = \frac{\sum_{i<j}(X_{ij} - \bar{x})(Y_{ij} - \bar{y})}{\sqrt{\sum_{i<j}(X_{ij} - \bar{x})^2 \sum_{i<j}(Y_{ij} - \bar{y})^2}}$$

where X_{ij} is the height of the node in the dendrogram where points i and j are first joined and $\bar{x}$ is the mean of all those heights. `scipy` also provides a built-in function that applies cophenetic correlation to dendrograms.

We don't expect perfect correlation between the dendrograms created by single-linkage and LSH for two reasons. First of all, the very point of LSH is that the minimum distances are approximate so it's always possible that close points don't get bucketed by the same hashes, in which case they may meet at a node unrealistically near the root. More importantly though, the height of each node isn't controlled by distance between points but by the minimum threshold distance r.

That being said the correlation is *near* perfect. Moreover, the correlation has an inverse relationship with the increase ratio: as A decreases, the correlation inches closer to 1. Since A can be loosely interpreted as a learning rate parameter, this relationship is intuitive — the slower we increase r, the more closely the algorithm approximates single linkage (or any alghorithm that considers more inter-point distances). Here we see the cophenetic correlation as calculated for three values of A:

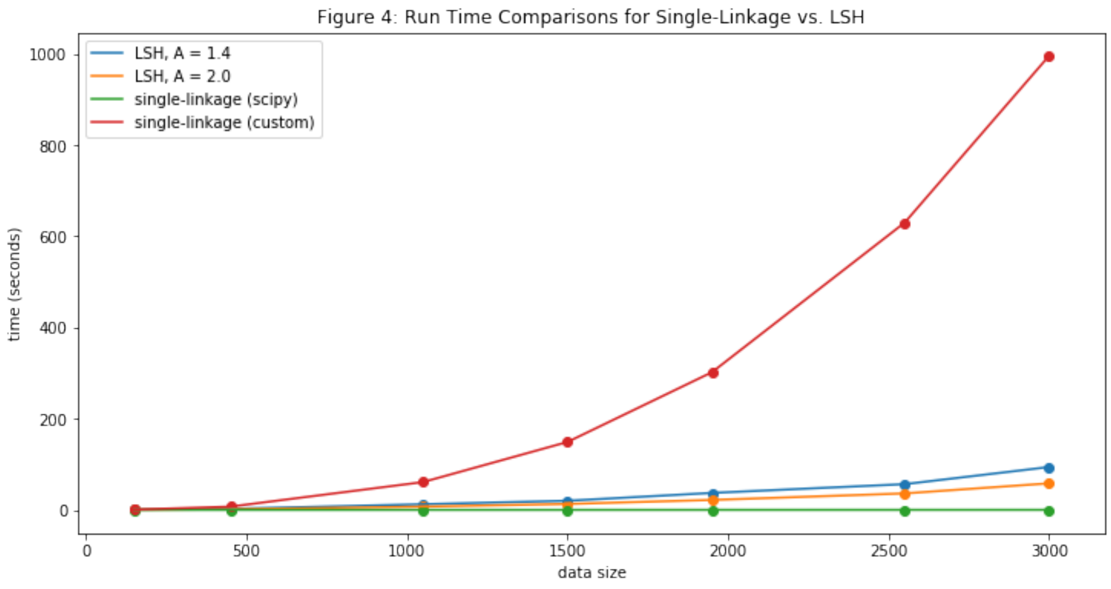
```
In [8]: Z1 = linkage(iris, method="single")
        clusters2, Z2 = lsh.LSHLink(iris, A = 2.0, l = 10, k = 100,
                                    dendrogram = True, seed1 = 12, seed2 = 6)
        clusters3, Z3 = lsh.LSHLink(iris, A = 1.4, l = 10, k = 100,
                                    dendrogram = True, seed1 = 12, seed2 = 6)
        clusters4, Z4 = lsh.LSHLink(iris, A = 1.2, l = 10, k = 100,
                                    dendrogram = True, seed1 = 12, seed2 = 6)

        C1 = cophenet(Z1)
        C2 = cophenet(Z2)
        C3 = cophenet(Z3)
        C4 = cophenet(Z4)
        print(np.corrcoef(C1, C2)[0,1])
        print(np.corrcoef(C1, C3)[0,1])
        print(np.corrcoef(C1, C4)[0,1])
```

```
0.9904974707929263
0.9987309030621808
0.9992070012882861
```

In addition to analyzing accuracy between single-linkage and LSH, we must look at performance. To some extent, it is possible to control the run time for LSH by changing the initial parameters, particularly A. Setting a larger starting value for A will result in a coarser but quicker run, while a smaller value will provide a slower and finer approximation. The size of the data also plays a significant role. For small datasets, single-linkage can be equivalent to or even outperform LSH. This is due to the fact that LSH has non-trivial overhead in computing the hash tables for each iteration; when there are not many data points, it can be slower to calculate the hash values on each iteration than to just do $n - 1$ merges as in single-linkage. As the dataset grows however, LSH scales linearly ($O(n)$) while single-linkage scales quadratically ($O(n^2)$). This is when LSH becomes particularly useful, as long as a high degree of precision is not needed.

Below, we show the run time in seconds on an enlarged *iris* dataset (with noise added as mentioned above) on four different hierarchical clustering implementations: scipy's single-linkage, our implementation of single-linkage, LSH with $A = 1.4$, and LSH with $A = 2.0$. Due to prohibitive run times on our single-linkage implementation for large datasets, we have only shown up to 3,000 data points.

```
In [19]: funcs.iris_run_times()
```



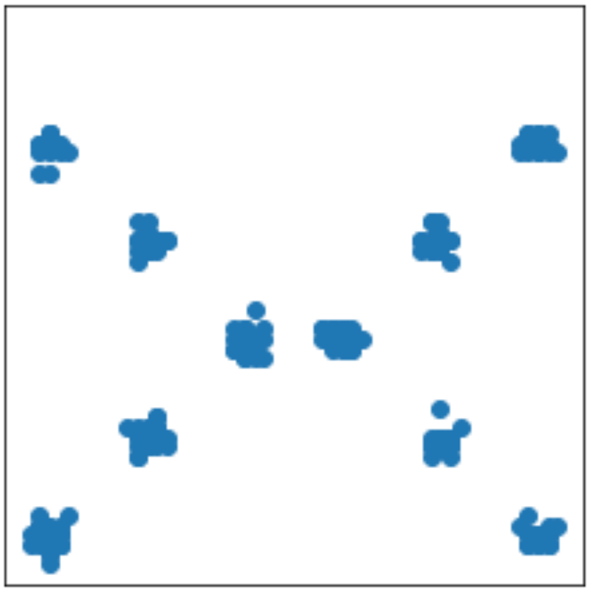
`scipy` is clearly orders of magnitude faster and is only shown above as an example of the theoretical possibility for LSH. For discussion purposes, we look at our custom implementation of single-linkage compared to LSH under the two different parameterizations $A = 1.4$ and $A = 2.0$. The single-linkage method clearly grows quadratically and performs relatively worse as the data size increases; on the other hand, LSH performs nearly linearly (we would expect that fully optimized code would have precisely linear run time). We observe that, although for small datasets LSH does not necessarily represent an improvement, the gains in efficiency for larger data can be substantial.

5.0.2 Synthetic Data of Abstract Shapes

We now illustrate our implementation of LSH for agglomerative hierarchical clustering on several generated datasets containing different shapes.

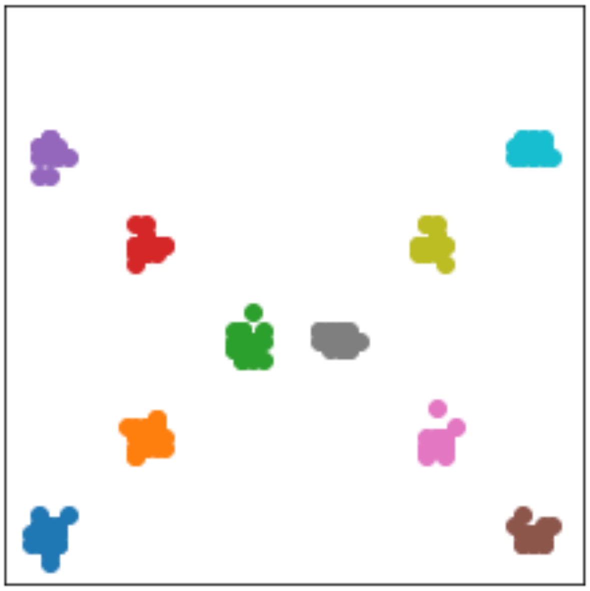
We start with a dataset of 200 data points, replicating synthetic data from the Koga et al. paper:

```
In [20]: tencircles = np.genfromtxt('../data/tencircles.csv', delimiter=",")
         lsh.plot_square(tencircles)
```



Running our implementation of LSH clustering on this data, with a cutoff of 10 clusters, yields the following:

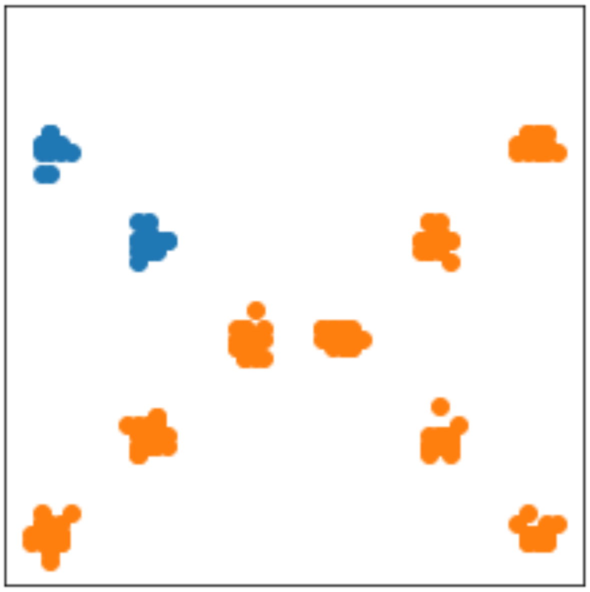
```
In [21]: lsh.plot_clusters(tencircles, cutoff=10, scale=10, linkage='LSH', A=1.4, k=10, l=100)
```



As clearly shown, the algorithm successfully identifies the ten clusters that are visible to the human eye. Intuitively, a researcher might also identify a way to break the data into two clusters:

the five groups on the left and the five groups on the right. The algorithm, however, fails to capture this:

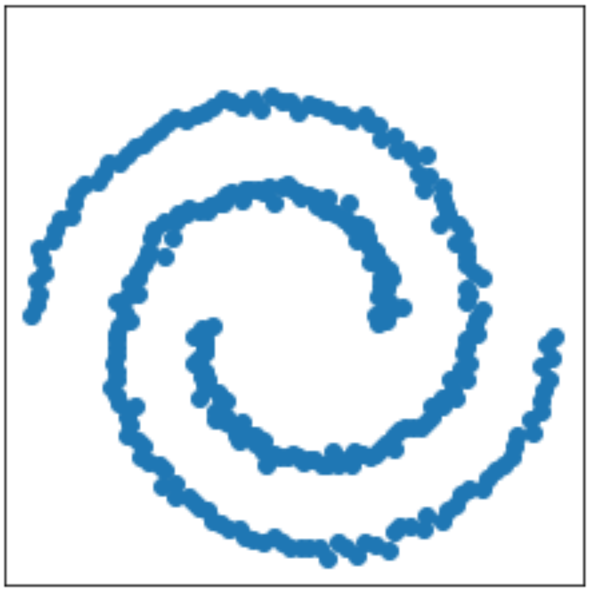
```
In [18]: lsh.plot_clusters(tencircles, cutoff=2, scale=10, linkage='LSH', A=1.2, k=10, l=100)
```



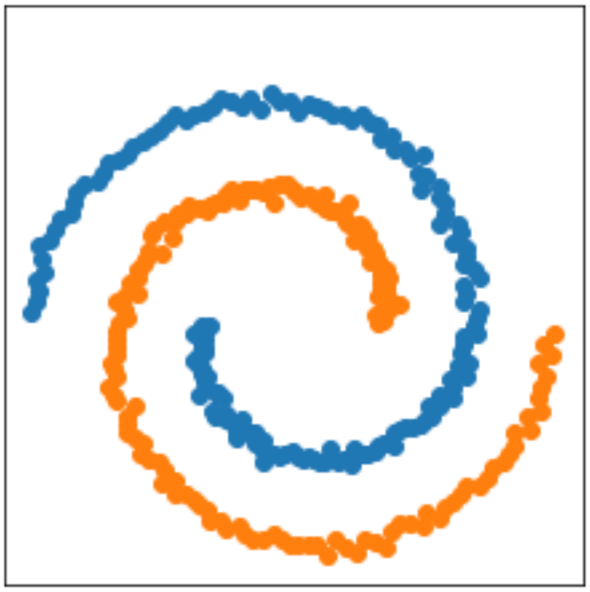
This could potentially be improved by changing some combination of the parameters set by the researcher, A, k, or ℓ. It is worth noting that the two center groups do appear closer to each other than does the center left with the two top left groups, so it is not unreasonable to expect that the algorithm would fail to recognize the pattern.

Now we turn to two other fabricated datasets in different shapes: the classic spiral that is often the canonical example for agglomerative hierarchical clustering (300 points) and a dataset with clusterings of different kinds of shapes (500 points). In both instances, the algorithm successfully identifies what we would perceive to be distinct clusters.

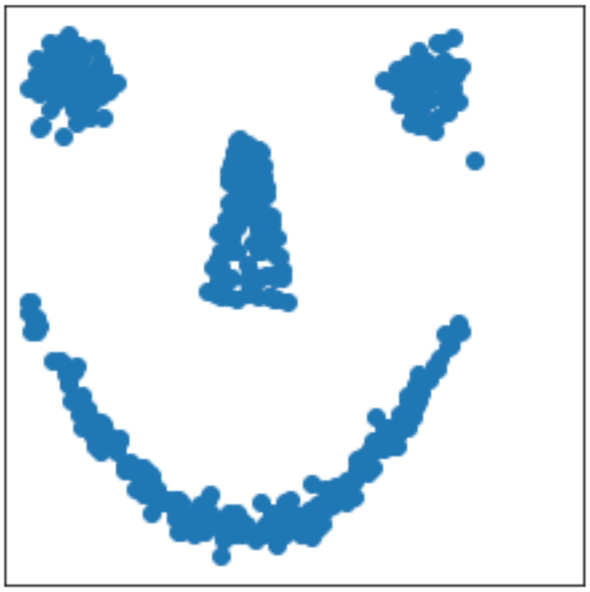
```
In [6]: spirals = np.genfromtxt('../data/spirals.csv', delimiter=",")[1:, 1:]
        lsh.plot_square(spirals)
```



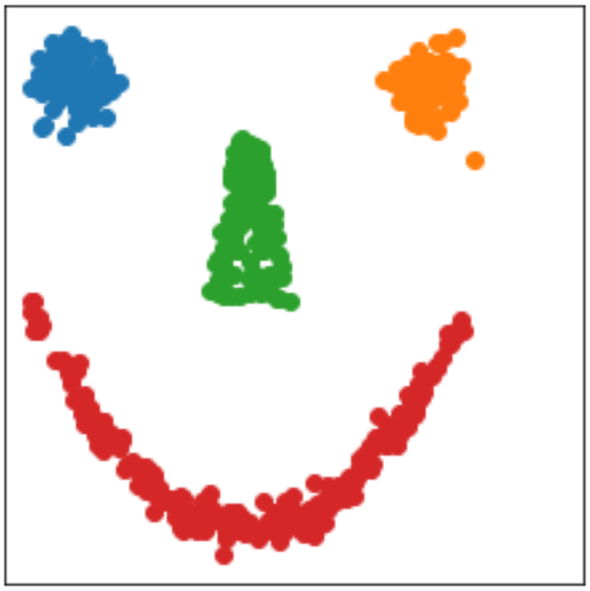
```
In [7]: lsh.plot_clusters(spirals, cutoff=2, scale=100, linkage='LSH', A=1.4, k=10, l=100)
```



```
In [10]: smiley = np.genfromtxt('../data/smiley.csv', delimiter=",")[1:, 1:]
         lsh.plot_square(smiley)
```



```
In [9]: lsh.plot_clusters(smiley, cutoff=4, scale=100, linkage='LSH', A=1.4, k=10, l=100)
```



6 Applications to Real Datasets

We apply LSH to the same real dataset that the paper uses, which is a sample of gene exprssion data collected from diabetic mice featuring 28 dimensions of cDNA. Again, we see that the dendrogram created by our clustering algorithm is not only similar to the one that single linkage yields, but that by decreasing the A paremeter to more slowly grow the threshold distance r, the results start to converge. In particular, an A value of 2.0 resulted in about 87% cophenetic correlation to the single linkage dendrogram, while an an A value of 1.2 resulted in nearly 91% cophenetic correlation.

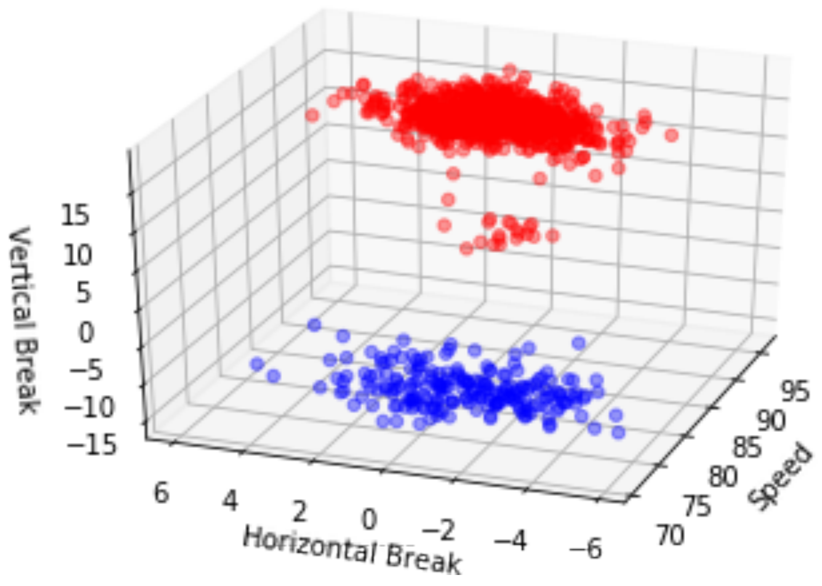
```
In [3]: gds10 = np.genfromtxt('../data/GDS10.csv', delimiter=",")[1:, 3:]
        gds = gds10[~np.isnan(gds10).any(axis=1)][1:501]

        Z1 = linkage(gds, method="single")
        _, Z2 = lsh.LSHLink(gds, A = 2.0, l = 40, k = 100,
                            dendrogram = True, seed1 = 12, seed2 = 6)
        _, Z3 = lsh.LSHLink(gds, A = 1.6, l = 40, k = 100,
                            dendrogram = True, seed1 = 12, seed2 = 6)
        _, Z4 = lsh.LSHLink(gds, A = 1.2, l = 40, k = 100,
                            dendrogram = True, seed1 = 12, seed2 = 6)

        C1 = cophenet(Z1)
        C2 = cophenet(Z2)
        C3 = cophenet(Z3)
        C4 = cophenet(Z4)

In [4]: print(np.corrcoef(C1, C2)[0,1])
        print(np.corrcoef(C1, C3)[0,1])
        print(np.corrcoef(C1, C4)[0,1])

0.8712429633263683
0.8853979368969566
0.9065300278917617
```

We turned to Major League Baseball (MLB) for a second real dataset that was not in the paper. The league has cameras installed in all of its stadium that capture the speed and trajectory of pitched balls. There are libraries available that let users download this so-called PITCHF/x data. For our purposes, we gathered 1,000 of Clayton Kershaw's pitches, filtering for the pitch types of either fastball or curveball. While the data has several dozen columns, for the sake of visualization we select three that we know to be important: the speed of the pitch (in miles per hour), the horizontal break (in inches), and the vertical break (in inches). We expect this data to cluster well because fastballs and curveballs are very distinct pitches, especially for a good player like Kershaw, with the former being faster (naturally) and having less break in either direction. Indeed, not only does the LSH algorithm break the clusters apart easily, but when comparing the classifications back to the original identities of the pitch types, we see 100% accuracy.

```
In [2]: kershaw = pd.read_csv('../data/Kershaw.csv').iloc[:, 1:5]
        kershaw_data = np.array(kershaw.iloc[:, 0:3])
        kershaw_labels = np.array(kershaw.iloc[:, 3])

        a = np.array([0])
        b = np.min(kershaw_data, axis = 0)[1:3]
        shift = np.r_[a, b]
        kershaw_data_shifted = kershaw_data - shift
        kersh_clusters = lsh.LSHLink(kershaw_data_shifted,
                                     A = 1.4, l = 10, k = 100,
                                     seed1 = 12, seed2 = 6, cutoff = 2)
```

```
In [3]: kersh_full = np.c_[kershaw_data, kersh_clusters]
```

```
In [4]: g1 = np.where(kersh_clusters == np.unique(kersh_clusters)[0])[0]
        g2 = np.where(kersh_clusters == np.unique(kersh_clusters)[1])[0]

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(kersh_full[g1, 0], kersh_full[g1, 1], kersh_full[g1, 2],
                   alpha = 0.4, c = 'red')
        ax.scatter(kersh_full[g2, 0], kersh_full[g2, 1], kersh_full[g2, 2],
                   alpha = 0.4, c = 'blue')
        ax.view_init(azim=200)
        ax.set_xlabel('Speed')
        ax.set_ylabel('Horizontal Break')
        ax.set_zlabel('Vertical Break')
        plt.show()
```



```
In [5]: class_LSH = kersh_clusters == 32
        class_TRUE = kershaw_labels == 'FF'
        sum(np.equal(class_LSH, class_TRUE))/kershaw.shape[0]
```

```
Out[5]: 1.0
```

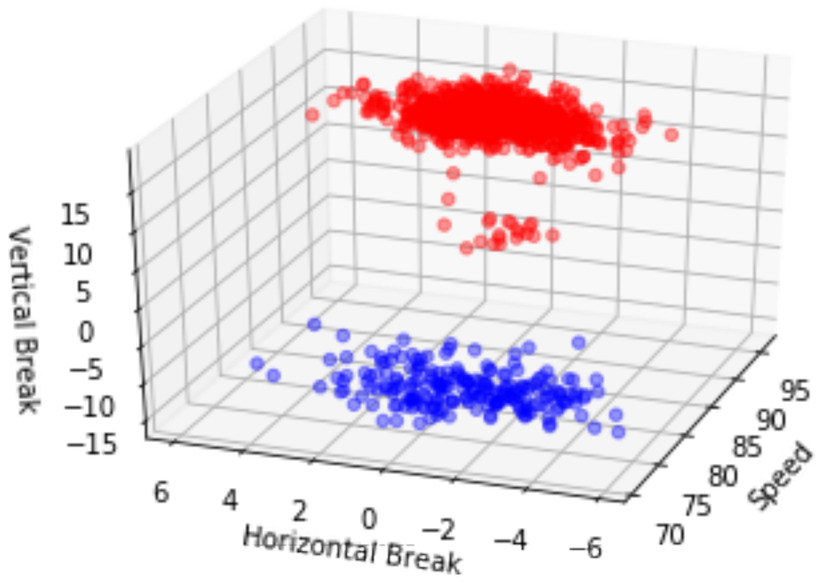
7 Comparative Analysis with Competing Algorithms

Another popular clustering algorithm is k-means, which, given a number of clusters, randomly initializes that many cluster centers and then iteratively adjusts their positions while classifying points based on the closest center until no points are being reclassified. We wrote a k-means clustering algorithm from scratch thats performs equally well to our LSH algorithm on the fastball/curveball data from above:

```
In [9]: np.random.seed(1)
        clust, cen = funcs.kmeans(2, kershaw_data)
```

```
iteration: 1
number of updates: 448
iteration: 2
number of updates: 215
iteration: 3
number of updates: 19
iteration: 4
number of updates: 1
iteration: 5
number of updates: 0
```

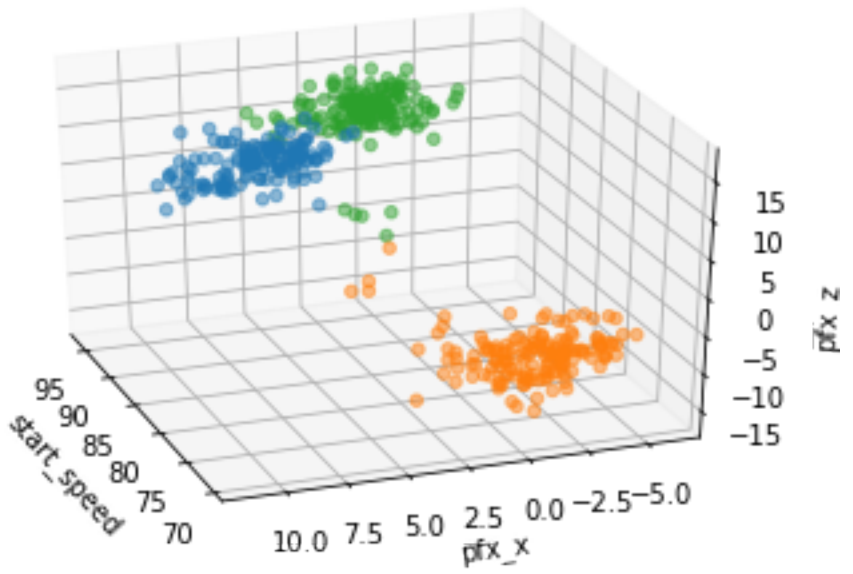
```
In [11]: kersh_full = np.c_[kershaw_data, clust]
         g1 = np.where(kersh_clusters == np.unique(kersh_clusters)[0])[0]
         g2 = np.where(kersh_clusters == np.unique(kersh_clusters)[1])[0]

         fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
         ax.scatter(kersh_full[g1, 0], kersh_full[g1, 1], kersh_full[g1, 2],
                    alpha = 0.4, c = 'red')
         ax.scatter(kersh_full[g2, 0], kersh_full[g2, 1], kersh_full[g2, 2],
                    alpha = 0.4, c = 'blue')
         ax.view_init(azim=200)
         ax.set_xlabel('Speed')
         ax.set_ylabel('Horizontal Break')
         ax.set_zlabel('Vertical Break')
         plt.show()
```



However, k-means does have a comparative advantage over both single linkage and LSH clustering when we introduce a third pitch, a change-up, that from a speed and break perspective, is somewhere between a fastball and curveball. The two aforementioned algorithms struggle to find three unique clusters in this data since they work from point to point instead of from inside out via cluster centers, so overlapping boundary regions between pitches prove confusing. However, k-means easily separates the three:

```
In [14]: kershawFCC = pd.read_csv('../data/KershawFCC.csv').iloc[:, 1:5]
         kershawFCC_data = np.array(kershawFCC.iloc[:, 0:3])
         kershawFCC_labels = np.array(kershawFCC.iloc[:, 3])

In [15]: np.random.seed(1)
         FCCclust, FCCcen = funcs.kmeans(3, kershawFCC_data)

iteration: 1
number of updates: 302
iteration: 2
number of updates: 3
iteration: 3
number of updates: 0


In [16]: kersh_full = np.c_[kershawFCC_data, FCCclust]
         g1 = np.where(FCCclust == np.unique(FCCclust)[0])[0]
         g2 = np.where(FCCclust == np.unique(FCCclust)[1])[0]
         g3 = np.where(FCCclust == np.unique(FCCclust)[2])[0]

In [17]: fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
```

```
ax.scatter(kersh_full[g1, 0], kersh_full[g1, 1], kersh_full[g1, 2], alpha = 0.5)
ax.scatter(kersh_full[g2, 0], kersh_full[g2, 1], kersh_full[g2, 2], alpha = 0.5)
ax.scatter(kersh_full[g3, 0], kersh_full[g3, 1], kersh_full[g3, 2], alpha = 0.5)
ax.view_init(azim=160)
ax.set_xlabel('start_speed')
ax.set_ylabel('pfx_x')
ax.set_zlabel('pfx_z')
plt.show()
```



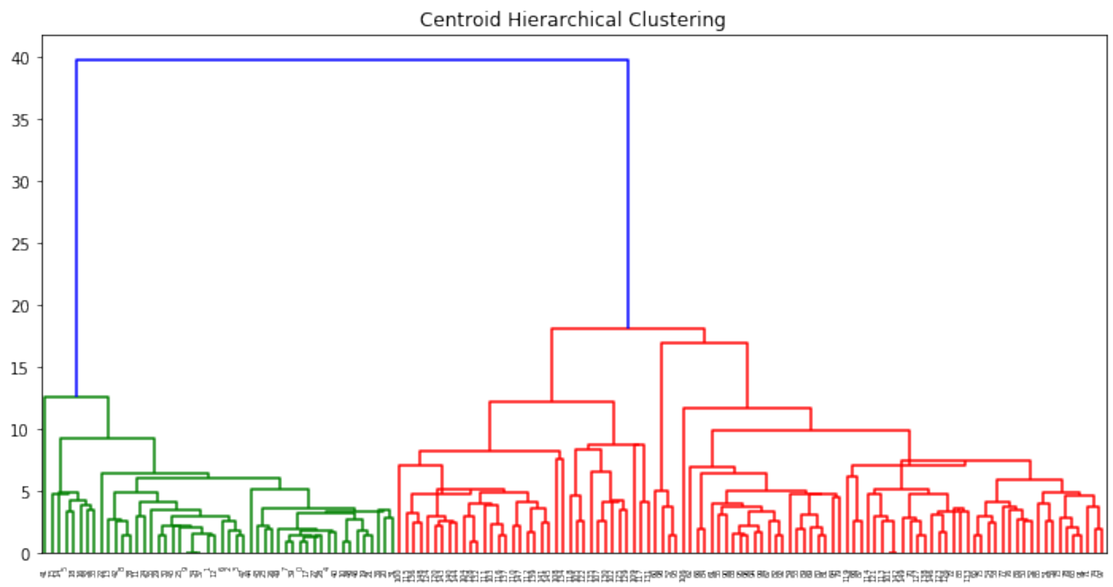
That being said, k-means struggles with non-spherical clusters where classes don't necessarily have similar distances from a central point like the synthetic spiral data shown above.

An interesting hybrid between single linkage hierarchical clustering and k-means clustering is centroid linkage hierarchical clustering. The algorithm starts by making each individual point the center of its own class, and then iteratively combines and recalculates centroids based on shortest distance. Like single linkage, it successively merges points, but like k-means the classifications are defined by a centroid that averages over the members of its class.

This method, not surprisingly, produces similar results to our LSH algorithm, as demonstrated by the 97.3% cophenetic correlation between the dendrograms. An interesting note about the centroid algorithm worth mentioning is that due to the nature of the algorithm, the dendrogram nodes don't necessarily have to unilaterally increase in height. If you look closely at the dendrogram you can see several examples of a merge between nodes ocurring below the heights of the two children.

This is because when classes merge, the centroid of the newly created group is recalculated as an average of the coordinates of the points that make it up. Therefore, it's possible that the centroid could be closer to another centroid than were either of its children. As a result, you see in the dendrogram that some merges grow down (slightly) instead of up. This means that the usual feature of being able to cut a dendrogram at any height to reveal the clusters at that juncture does not apply to centroid linkage.

```
In [97]: Z1 = linkage(iris, method="centroid")
         dendrogram(z)
         plt.gcf().set_size_inches(12, 6)
         plt.title('Centroid Hierarchical Clustering')
         plt.show();
```



```
In [98]: clusters2, Z2 = lsh.LSHLink(iris, A = 1.4, l = 10, k = 100,
                                     dendrogram = True, seed1 = 12, seed2 = 6)
         C1 = cophenet(Z1)
         C2 = cophenet(Z2)
         print(np.corrcoef(C1, C2)[0,1])

0.9727173842543353
```

8 Discussion / Conclusion

Unsupervised learning, and clustering in particular, are currently hot topics in the world of machine learning and data science because they have so many applications, from bioinformatics to customer segmentation. With the propagation of high-dimensional data, efficient versions of clustering algorithms become of utmost importance, since the difference between linear and quadratic runtime for large datasets can separate theoretical use from practical implementations. For those reasons, locality-sensitive hashing for agglomerative hierarchical clustering certainly fulfills a need in the realm of statistical computing.

An extension of the algorithm that wasn't mentioned in the original paper might consider how to apply it to on wider varieties of datasets. In other words, even though we specified various implementation details to deal with cases of negative numbers or rounding, how to run the algorithm on data with categorical or factor columns was never mentioned. Would one-hot encoding solve the problem or are more rigorous transformations necessary? Having a more dynamic version of the algorithm that could parse data of less homogenous types would allow it to be applied to more datasets, and most likely open doors for new domains.

And finally, one of the limitations (that is by no means uncommon in machine learning) of the algorithm was how much manual and ad-hoc tuning of parameters was required to find optimal solutions. While some guidance was given in the original paper, applying the algorithm to new datasets requires lots of fiddling around and various initializations of the parameters; and given how many there are, this can be quite time-consuming. Further research might seek to establish rules of thumb in terms of tuning.

9 References / Bibliography

1. Eaves IA, Wicker LS, Ghandour G, Lyons PA et al. Combining mouse congenic strains and microarray gene expression analyses to study a complex trait: the NOD model of type 1 diabetes. Genome Res 2002 Feb;12(2):232-43. PMID: 11827943
2. Koga H, Ishibashi T, Watanabe T (2006) Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing. In: Knowledge and Information Systems 12(1): 25-53
3. Sievert C (2015). pitchRx: Tools for Harnessing 'MLBAM' 'Gameday' Data and Visualizing 'pitchfx'. R package version 1.8.2.

10 Appendix

```
In [27]: n, d = iris.shape
         clusters = np.arange(n)
         C = int(np.ceil(np.max(iris))) + 1
         k = 10

In [28]: ### increase l

In [29]: l = 10
         args = []
         for i in range(l):
             args.append((C, k, iris))

In [17]: # l = 10
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

162 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
256 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: l = 20
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [14]: # l = 20
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

291 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
423 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: l = 30
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [42]: # l = 30
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

426 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
641 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: l = 40
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [39]: # l = 40
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

586 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
993 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: l = 50
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [21]: # l = 50
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

703 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
1.02 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: l = 100
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [24]: # l = 100
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

1.42 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
2.01 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: l = 200
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [28]: # l = 200
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

2.83 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
4.14 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: l = 500
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [32]: # l = 500
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

7.23 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
10.9 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: ### increase n

In [31]: k = 10
         l = 10

In [32]: iris = datasets.load_iris().data
         iris = v1.data_extend(iris, 1) * 10
         iris += np.abs(np.min(iris))
         n, d = iris.shape
         clusters = np.arange(n)
         C = int(np.ceil(np.max(iris))) + 1
         args = []
         for i in range(l):
             args.append((C, k, iris))
```

```
In [77]: # n = 150
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

181 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
257 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 2) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [57]: # n = 300
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

391 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
530 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 3) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [61]: # n = 450
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

440 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
624 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 4) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [104]: # n = 600
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

577 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
821 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 5) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [108]: # n = 750
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

925 ms ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
1.06 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 7) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [85]: # n = 1050
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

1.1 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
1.45 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 10) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [89]: # n = 1500
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

1.83 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
2.23 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 14) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [98]: # n = 2100
         %timeit -r1 funcs.mp(v1.build_hash_table, args)
         %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)

2.05 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
2.96 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)


In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 33) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))

In [115]: # n = 4950
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)
```

```
6.81 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
8.93 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 67) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [119]: # n = 10050
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)
```

```
13 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
18 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```

```
In [ ]: iris = datasets.load_iris().data
        iris = v1.data_extend(iris, 100) * 10
        iris += np.abs(np.min(iris))
        n, d = iris.shape
        clusters = np.arange(n)
        C = int(np.ceil(np.max(iris))) + 1
        args = []
        for i in range(l):
            args.append((C, k, iris))
```

```
In [125]: # n = 15000
          %timeit -r1 funcs.mp(v1.build_hash_table, args)
          %timeit -r1 v1.build_hash_tables(C, d, l, k, iris, clusters)
```

```
20 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
26.3 s ś 0 ns per loop (mean ś std. dev. of 1 run, 1 loop each)
```